一起学

koa

目錄

一起学koa

> 作者：17koa
>
> 来源：koa-generator-examples

koa是下一代基于nodejs的modern web framework，会用到很多es高级特性，可以说无论从es学习还是web开发都有必要学习。

鉴于目前学习资料不多，故有此书

年前几天比较忙，更新暂缓

宗旨

大家一起学习koa

- 暂时不会的可以学会
- 会的可以帮助他人，查缺补漏，提供更多最佳实践

参与流程

通过提问、实现，pr的方式

- 提issue
- 根据某个issue，fork并实现
- 提交pr
- 合并pr并提交
- 发布到git pages上

版本说明

目前Koa 2.x还没有发布，先以Koa 1.x为主

koajs 1.x和2.x的区别

1.x和2.x的都是基于ctx（上下文）模型实现的

目前2.x还没没有完全定下来

- nodejs 4.0+支持的es6语法
- async/await支持 (现在须借由 babel)
- generator不能直接使用，必须使用co类的包装后才可以

目录

预览

- 一起学koa之1.x版本预览
- 一起学koa之2.x版本预览

技术顾问

- @fundon
- @alsotang
- @老雷

用明白不一定写明白，但写明白就一定能用明白，大家加油

Contributing

1. Fork it
2. Create your feature branch (`git checkout -b my-new-feature`)
3. Commit your changes (`git commit -am 'Add some feature'`)
4. Push to the branch (`git push origin my-new-feature`)
5. Create new Pull Request

Version History

- v0.1.0 初始化版本

欢迎fork和反馈

如有建议或意见，请在issue提问或邮件

当然也可以在国内最专业的cnode论坛上回复《一起学koa》

License

this repo is released under the MIT License.

Nodejs 4.x

关于node各版本解释

最近node官网发布了更新日志 Node v5.1.1 (Stable), v4.2.3 “Argon” (LTS), v0.12.9 (LTS) and v0.10.41 (Maintenance) are released 请问这些版本有什么区别，适用于什么场景，我发现node官方提供下载的是 4.2.3和5.1.1，但yum 安装的是 0.10.41 网上搜了下没找到答案，只能靠cnode社区啦~多谢！

- Stable是当前稳定版本
- LTS是长期支持版本
- Maintenance是维护，不再增加任何feature，仅做重大bug修复

Why use 4.x?

目前4.2.*是LTS（长期支持版本），支持非常多的es6特性，这些都是koa的基石，可以让大家更好的利用es6开发modern web app。

新特性

Node.js 4.0.0 可以让您享受最尖端的技术，保持项目的先进性。其中对 v8 的升级几乎做到了与 Chromium / Google Chrome 同步，达到了 4.5.x，它提供了很多新的语言功能。ECMA-262 是 JavaScript 语言规范的最新版本，而且好多新特性数都是开箱即用的。这些新特性包括：

- classes - 各种 ‘类’，再也无需用 CoffeeScript 的语法糖写类了
- typed arrays — 类型数组
- generators - 未来的.js 代码中将有无数生成器，不学一点就看不懂 JS 代码了哦
- collections — 集合、映射、弱集合、弱映射
- arrow functions — 箭向函数
- block scoping — 使用 let 、const 作用域，块辖域
- template strings — 模板字串
- promises — 用标准化了的方法进行延迟和异步计算
- symbols — 唯一的、不可修改的数据

下面会逐一讲解

classes

在ES6中声明一个class

在ES6中，你可以使用如下的方式进行Class声明。在使用的过程中，有一点需要特别注意，一定要先使用下面的任何一种方式声明的**class**，才能引用**class**定义。这个和原先的JavaScript prototype方式声明有很大的区别，在原先的方式中，因为class是通过function来声明的，而在javascript中，function将自动变成全局可见，所以class的定义可以出现在引用之后。
在后面的文章中，为了保持代码的一致性和文章的清晰，我们将只适用第一种方式进行class定义

- 使用class关键字

```
class Polygon{
}
```

*使用class表达式进行声明

```
var Polygon = class {
}
```

定义class的构造函数

通过使用constructor关键字，你可以给class定义一个构造函数。不过在整个class的定义中，只能有一个构造函数存在.下面是一个具体的例子

```
class Polygon{
  constructor(height, width){
      this.height = height;
      this.width = width;
  }
}
```

给class增加成员属性

在ES6中， 你可以通过getter和setter方法，给类增加属性。如果属性只有getter方法，那么它就是一个只读属性；如果属性同时又setter和getter方法，那么它就是一个可读写属性。请看下面的例子
注意属性 `name` 和 `_name`
因为我们定义了 name 的读写器，而没有定义 _name 的读写器，所以访问这两个属性的结果是不同的。

```
class Person{
  constructor(name){
      this._name = name;
  }
  get name(){
      return this._name.toUpperCase();
  }
  /**
   * 注意一点，不要这样写：
   * set name(somename) {
   *  this.name = somename;
   * }
   * 因为给 this.name 赋值的时候会调用 set name ，这样会导致无限递归直
   *
   */
  set name(somename){
      this._name = somename;
  }
}
```

给class增加成员函数

这点没什么可说的，就是在类定义中增加函数定义即可，请看下面的例

```
class Polygon{
    constructor(height, width){
        this.height = height;
        this.width = width;
    }
    get name(){
        return this._name;
    }
    set name(somename){
        this._name = somename;
    }

    //readonly property
    get area(){
        return calcArea();
    }

    calcArea(){
        return this.height * this.width;
    }
}
```

实现class的继承

在ES6中，通过使用extends关键字，你可以使用类的继承

```
class Animal{
    constructor(name){
        this.name = name;
    }

    say(){
        console.log(this.name + " try to say something...");
    }
}
class Dog extends Animal{
    say(){
        //可以通过super关键字来调用父类的方法
        super.say();
        console.log(this.name + " barking...");
    }
}
```

静态成员函数

在ES6中，你可以使用static关键字来定义静态成员函数。静态成员函数的功能主要是对外提供一些辅助方法。在下面的例子中，Point类就向外提供了一个辅助方法来计算2点间的距离

```
class Point(){
    constructor(x, y){
        this.x = x;
        this.y = y;
    }
    static distance(a, b){
        constant dx = a.x - b.x;
        constant dy = a.y - b.y;
        return Math.sqrt(dx * dx + dy * dy);
    }
}
```

typed arrays

类型数组

概述

Generator是ES6引入的实现异步操作的一种新方法，在Generator出现之前，不管哪种方法，异步操作都是使用回调函数来实现的。只从出现了Generator之后，开发人员可以使用同步调用的逻辑来实现异步操作，只要在需要等待的地方，使用yield语句放弃运行即可。

Generator的基本写法

和学习所有其他语言特性一样，我们使用一个Hello World来作为Generator的入门代码

```
function* helloworld(){
    yield "Hello";
    return "World!";
}

func = helloworld();
func.next();//return { value: 'Hello', done: false }
func.next();//return { value: 'World!', done: true }
func.next();//return { value: '', done: true }
```

从上面的代码的输出可以看出

- Generator函数的定义，是通过**function** *实现的
- 对Generator函数的调用返回的实际是一个遍历器，随后代码通过使用遍历器的next方法来获得函数的输出
- 通过使用**yield**语句来中断Generator函数的运行，并且可以返回一个中间结果
- 每次调用**next**方法，Generator函数将执行到下一个yield语句或者是return语句。下面我们就对上面代码的每次next调用进行一个详细的解释
    - 第一次调用next方法的时候，函数执行到 `yield "Hello"` 语句停了下来，并且返回了Hello这个value，随同value返回的done属性表明Generator函数的运行还没有结束
    - 第二次调用next方法的时候，函数执行到 `return "World!"` 语句停了下来，并且返回了World!这个value，随同value返回的done属性表明Generator函数的运行已经结束
    - 第三次调用next方法的时候，由于Generator函数执行已经结束了，所以函数调用立即返回，done属性表明Generator函数已经结束运行，value是空的，因为这次调用并没有执行任何语句

yield语句

yield语句在Generator函数的执行过程中扮演了中断/暂停执行函数的功能。每次你调用next()方法的时候，Generator函数都将执行到下一个yield语句或者return语句，当执行到yield语句的时候，如果yield语句跟着一个表达式，那么表达式的值将

作为value被返回。

next方法参数

由于yield语句只是抛出value, 但是本身并不返回value，如果你要yield语句有返回值，就要在调用next方法的时候，传入一个参数，这个参数就将作为上一个yield语句的返回值，下面是一个例子

```
function* f() {
  for(var i=0; true; i++) {
    var reset = yield i;
    if(reset) { i = -1; }
  }
}

var g = f();

g.next() // { value: 0, done: false }
g.next() // { value: 1, done: false }
g.next(true) // { value: 0, done: false }
```

for...of循环

在上面的所有例子中，如果我们要让Generator函数执行下一步，就必须调用一次next方法，那么有没有什么方法让Generator函数自动执行每一步而不需要代码干预呢?答案是肯定的，我们可以使用for...of循环来实现，下面是一个具体的例子。在下面的例子中，for..of loop将遍历所有的yield语句（记住只是遍历**yield**语句，因此**return**语句的返回值是不会输出的）

```
function* loopThroughInt(){
    for (let index = 0; index < 100; index++){
        yield index;
    }
    return 100;
}
for (let v of loopThroughInt()){
    console.log(v); //output 0...99
}
```

yield*

如果在Generator函数内部需要调用另外一个Generator函数，那么对目标函数的调用就需要使用yield*，以下是一个简单的例子

```
function* objects(){
    yield "cat";
    yield "dog";
    yield "duck";
}
function* say(){
    yield* objects();
    yield " say hello world!";
}
```

在这个章节中，我们将介绍ES6中引入的集合对象和映射对象。因为涉及的内容比较多，所以我们将分成2个部分，分别对集合和映射对象进行介绍。

集合

集合对象类似于以前JavaScript中的数组，只是集合对象中不能存在相同的对象。

集合的初始化

集合对象是通过构造函数进行初始化的，调用格式有2种

- 不传参数给构造函数，这将创建一个空的集合对象

```
var sampleSet = new Set();
console.log(sampleSet.size); //will output 0
```

- 传递一个数组对象给构造函数，将创建一个包含这个数组中不重复内容的集合

```
var sampleSet = new Set([1,2,3,4,5,5,6]);
console.log(sampleSet.size);//will output 6
console.log(sampleSet);//will ouput Set {1,2,3,4,5,6}
```

向集合添加内容

可以通过使用集合对象的add方法向已经创建的集合添加新的内容

```
var sampleSet = new Set();
sampleSet.add(2);
sampleSet.add('2');
console.log(sampleSet.size); //will output 2
console.log(sampleSet); //will output Set { 2, '2' }

//例子2
var sampleSet = new Set();
sampleSet.add(NaN);
sampleSet.add(NaN);
console.log(sampleSet); //will output Set { NaN }
```

从上面程序的输出，可以看出

- 集合可以保存不同类型的数据
- 在向集合添加数据的时候，JavaScript并不进行数据转化。因此2和'2'是不一样的
- 对集合对象来说，所有的NaN都是一样的

向集合添加对象

我们之所以将向集合添加对象单独出来解释，是因为对对象是否相同的判断和普通对象不一样。下面是几个例子。

```
//例子1
var sampleSet = new Set();
var a = {};
var a1 = a;
//因为a和a1在底层指向的是同一个内存对象，所以a === a1
sampleSet.add(a);
sampleSet.add(a1);
console.log(sampleSet); //will output Set { {} }

//例子2
var sampleSet = new Set();
var a = {};
var a1 = {};
//因为a和a1在底层指向的是不同的内存对象，所以a != a1
sampleSet.add(a);
sampleSet.add(a1);
console.log(sampleSet); //will output Set Set { {}, {} }
```

集合的其他操作

除了向已经存在的集合添加内容，你还可以通过集合的成员函数实施下面的操作

- 删除内容，通过 `delete(value)` 来删除集合中的某个内容
- 判断集合是否包含内容，通过 `has(value)` 来判断集合中是否包含参数所指定的内容
- 清除所有内容，通过 `clear()` 你可以删除一个集合中的所有内容

集合对象的遍历

遍历集合对象最简单的方法就是遍历 `values()` 方法的返回值，下面是一个具体的例子

```
'use strict';
var sampleSet = new Set([1,2,3,4,5]);
//output is
//1
//2
//3
//4
//5
for (let val of sampleSet.values()){
  console.log(val);
}
```

遍历集合的第二个方法是条用 `forEach` 函数完成，下面是具体的例子。对于这个例子，大家一定很奇怪，为什么forEach接受的函数的参数是2个。对于这点我一开始也觉得很奇怪，在看了相关的文档才明白，实际上对于集合中的每个元素，都有2个属性与之绑定（Key和value)，只不过他们完全相同而已，我个人估计是为了和映射对象保持类似的结构，从而简化Javascript引擎而这么做的。

```
var sampleSet = new Set([1,2,3,4,5]);
//output is
//1
//2
//3
//4
//5
//实际上你用console.log(key)也会得到完全一样的结果
sampleSet.forEach((value, key) => {console.log(value);});
```

弱集合

弱集合对象和普通的集合对象有着下面这些不同点:

- 弱集合对象只能保存对象，不能保存普通数据
- 弱集合对象对另外一个对象的引用，并不影响垃圾回收器的工作。换句话说，如果一个弱集合 `set` 包含一个对象 `a` 的引用，如果在运行过程中，除了这个应用，没有任何其他地方应用 `a` 这个对象了，那么垃圾回收器将回收 `a` 。
- 弱集合对象没有 `size` 属性，也不能进行遍历。不能进行遍历的原因就是因为第二点。因为在遍历的过程中，集合中的对象随时可能被垃圾回收器给回收。

可以执行的操作

根据上面的描述，对弱集合对象可以进行的操作仅限于。

- 增加成员，使用 `add()` 方法增加成员
- 删除成员, 使用 `delete()` 方法删除成员

- 判断成员是否在集合，使用 `has()` 方法判断一个成员是否存在

弱集合的使用

弱集合的一个使用场合是你需要操作DOM的一个集合，但是你不想因为集合拥有对DOM对象的引用而阻止对DOM对象的自动销毁。

映射

我们可以把映射看成是对JavaScript Object的一个扩展。在传统的JavaScript Object中，当我们设置key, value对的时候，key只能是字符串。ES6对这个进行了扩展，形成了新的映射类型，映射类型的key, value对可以是任意对象。

映射的初始化

在ES6中，可以通过如下2种对构造函数的调用来创建并初始化一个映射对象

- 不传参数给构造函数，这将创建一个空的映射

```
var sampleMap = new Map();
console.log(sampleMap.size); //will output 0
```

- 传一个数组给构造函数，这个传入的数组的每个元素包含2个值，第一个将作为key, 另外一个作为value

```
var sampleMap = new Map([['key1', 'val1'], ['key2', 'val2']]);
//will output 2, also sampleMap will have key key1 and key2
console.log(sampleMap.size);//will output 2
```

对映射对象的操作

和集合对象类似，你可以对映射对象执行如下的操作

- 增加成员，通过 `set(key, val)` 向映射对象增加成员
- 删除成员，通过 `delete(key)` 从映射对象删除成员
- 测试是否包含，通过 `has(key)` 测试是否包含某个key
- 获取成员，通过 `get(key)` 返回成员
- 清空成员，通过 `clear()` 操作清空整个映射对象

另外，映射对象和集合对象一样，当你使用object作为key的时候，是根据key是否指向同样的内存对象来判断是否new一个key的。如果2个key的对象指向的是同一个内存对象，那么后面一个set将覆盖前面一个set的内容。

对映射对象的遍历

对映射对象，你可以使用下面的任意一种方法进行合适的遍历。

````
```javascript
'use strict';
var sampleMap = new Map();
sampleMap.set('key1', 'val1');
sampleMap.set('key2', 'val2');
for (let key of sampleMap.keys()){
console.log(key + '---' + sampleMap.get(key));
}
````

````
```javascript
'use strict';
var sampleMap = new Map();
sampleMap.set('key1', 'val1');
sampleMap.set('key2', 'val2');
for (let val of sampleMap.values()){
console.log(val);
}
````

````
```javascript
'use strict';
var sampleMap = new Map();
sampleMap.set('key1', 'val1');
sampleMap.set('key2', 'val2');
for (let item of sampleMap.entries()){
console.log(item[0] + '---' + item[1]);
}
````

* `forEach()` 调用

```
'use strict';
var sampleMap = new Map();
sampleMap.set('key1', 'val1');
sampleMap.set('key2', 'val2');
sampleMap.forEach(function(value, key, map){
  console.log(key + '---' + value);
});
```

弱映射

和弱集合一样，弱映射的key不被垃圾回收所检查，当key对应的对象被回收的时候，也自动从弱映射中被删除。由于这个原因，弱映射和弱集合一样无法进行遍历，这能修改。
```

arrow functions

箭向函数

Let命令

定义Let变量

在ES6中引入了let命令，通过let命令定义的变量只能在let命令所在的代码块内部被引用。

```
"use strict"
{
    let hello = "Hello World!"
}
console.log(hello) //会报错，因为没有全局的hello变量被定义
```

从上面的例子可以看出，使用Let命令定义的变量不会自动被提升为全局变量。相反的情况，如果在上面的例子中，你使用var来定义hello这个变量，那么hello这个变量将自动被提升为全局变量，就可以被后面的console.log访问了。

变量死区

ES6中规定，如果你在一个代码区块中使用了let命令来定义变量，那么在变量被定义之前，不允许对这个变量的访问存在。因此在let命令之前的所有代码被称为变量死区。如果在变量死区发生了对变量的引用，那么JavaScript引擎将报错。

```
"use strict"
{
    console.log(hello); //将报错，因为变量声明在之后
    let hello = "Hello World!";
}

#正确代码
{
   let hello = "Hello World!";
   console.log(hello);
}
```

不能重复声明变量

在同一个代码区块中，你不能使用let/var命令重复定义一个已经已经存在的变量。因此下面的代码将报错。

```
"use strict"
{
   let hello = "Hello World!";
   let hello = "Hello Not World!";//报错
   console.log(hello);
}
{
   let hello = "Hello World!";
   var hello = "Hello Not World!";//报错
   console.log(hello);
}
```

块级作用域

ES6引入let命令的一个副作用就是引入了块级作用域。让我们来看一个具体的例子来详细分析块级作用域。

```
function f() { console.log('I am outside!'); }
(function () {
  if(false) {
    // 重复声明一次函数f
    function f() { console.log('I am inside!'); }
  }

  f();
}());
```

在ES5环境中，这段代码将输出"I am inside!"，这个是因为在ES5中，不管代码块是否被运行，函数定义都将自动被提升到外部（全局空间）。但是如果在ES6中运行，你将看到输出"I am outside!"，这个是因为重复的function定义是在另外一个不被执行的代码块中。实际上在ES6中，上面的代码被翻译成了下面的ES5代码。

```
"use strict";

function sayHello() {
  console.log("say hello from global");
}

(function () {
  if (false) {
    var _sayHello = function _sayHello() {
      console.log("say hello from inside");
    };
  }

  sayHello();
})();
```

块级作用域的另外一个影响就是把变量绑定到了当前的作用域

```
 "use strict";
function hello() {
     let word = "hello world!";
     if (false) {
          let word = "hello world1!";
     }
     console.log(word);
}

hello();//will see "Hello World!"
```

const命令

const命令和let命令唯一的区别在于constant命令定义的是一个常量。因此使用constant命令定义的常量必须在定义的同时被初始化。

引入其他js文件中定义的变量/常量

在ES6中，可以使用import语句来引入其他文件中定义的常量 / 变量。

```
"use strict"
//定义在constants.js的常量
export const HELLO = "hello"

//在其他js中
import * as constants from "./constants.js"
console.log(constants.HELLO)

import {HELLO as myHello} from "./constants.js"
console.log(myHello)
```

全局变量的属性

在ES6中，使用var定义的变量将是全局变量的属性（window --- 在browser中 or global--在server段代码中)，因此可以通过window.或者global.来访问。但是在顶层代码中使用let / constant定义的变量/常量将不是全局变量的属性。

概述

在ES6引入模板字符串之前，如果大家需要在代码中创建一个包含变量的字符串，那么代码将非常难读，并且也非常容易出错。下面就是一个简单的例子，在例子中我们将输入的3个参数拼接在一起，然后返回给调用方。

```
//在模板字符串出现前的写法，写法冗长而且难于理解
function returnSomthing(param1, param2, param3){
    return "something return based on input("
           + "param1:" + param1.toString() + "---"
           + "param2:" + param2.toString() + "---"
           + "param3:" + param3.toString();
}

//使用模板字符串的写法,
function returnSomthingNew(param1, param2, param3){
    return `something return based on input(
              param1:${param1}---param2:${param2}---param3:${param3
}
```

通过上面的代码，你可以看出在使用模板字符串之后，代码变得非常简洁，而且也容易阅读。下面是在使用模板字符串时候的一些注意点

- 模板字符串是使用 ` 引用起来的，如果在最终生成的字符串中包含`字符，那么需要使用\字符进行转义
- 模板字符串中对于变量的引用是通过${}来进行的
- 使用模板字符串的时候，${}中可以放入任意合法的JavaScript表达式。JavaScript对包含在${}中的内容实际上是通过eval表达式来进行的

标签模板

模板字符串可以跟在一个函数名之后，该函数将被调用来处理跟在后面的模板字符串，这个功能被称为标签模板。被调用的函数将接收到下面的参数列表(literals,...values)。其中literals是一个数组，内容是模板字符串中不需要进行变量替换的部分，而values就是每个替换变量经过eval之后的值，下面是一个具体的例子。

```
var total = 30;
var msg = transform`The total number is ${total}`;
total = 20;
var msg1 = transform`The total number is ${total}`;
//in our sample
//literals = ["The total number is ", ""]
//values = [30]
function transform(literals,...values){
    var output = "";
    for (var index = 0; index < values.length; index++){
        if (parseInt(values[index]) >= 30){
            output += literals[index] + "high value";
        }else{
            output += literals[index] + "low value";
        }
    }
    output += literals[index];
    return output;
}
console.log(msg); //output The total number is high value
console.log(msg1);//output The total number is low value
```

Promise的历史

在Promise出现之前，如果大家需要实现异步操作，通用的做法是事件加上回调函数，如果我们有多个异步操作需要嵌套执行的话，那么代码将变得非常难于阅读。让我们来看一个具体的代码例子。在下面的例子中，我们首先通过Http request调用一个web service，然后将从web service收到的数据写入一个本地文件。从任务的角度来看，这是一个非常简单的任务，但是当你第一次看到这个代码的时候，一定觉得头很晕，因为在这段代码中

- 对web service进行调用的代码和写文件的代码混杂在一起（写文件的代码嵌套在 `end` 事件的回调函数中），造成代码模块不清晰，阅读和理解起来比较费劲。
- 对错误的处理分散在代码的各个地方，而且错误处理的实现方式都不一样。对web service进行调用的代码，通过监听 `error` 事件来处理错误，并且将错误输出到控制套；而写文件的代码，是通过回调函数来处理错误，并将错误通过 `throw` 语句抛出。

```
var http = require("http");
var fs = require("fs");
var querystring = require("querystring");
var postData = querystring.stringify({
  'msg' : 'Hello World!'
});

var options = {
  hostname: '127.0.0.1',
  port: 8080,
  path: '/upload',
  method: 'POST',
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded',
    'Content-Length': postData.length
  }
};

var req = http.request(options, (res) => {
  console.log(`STATUS: ${res.statusCode}`);
  console.log(`HEADERS: ${JSON.stringify(res.headers)}`);
  res.setEncoding('utf8');

  var dataReceived = ""

  res.on('data', (chunk) => {
    dataReceived = dataReceived + chunk.toString();
    console.log(`BODY: ${chunk}`);
  });

  res.on('end', () => {
    console.log('No more data in response.')
    //now we try to write the message to a file
    fs.writeFile("temp.txt", dataReceived, function(err){
        if (err){
            throw err;
        }
        console.log("Write file temp.txt succ");
    });
  })
});

req.on('error', (e) => {
  console.log(`problem with request: ${e.message}`);
});

// write data to request body
req.write(postData);
req.end();
```

接下来，让我们使用Promise重写上面的代码。在重写的代码中，我们可以看到：

- 对web service进行调用的代码和写文件的代码完全分离开了，代码结构变得非常清晰。在阅读代码的过程中，不会再被不相关的代码所干扰。
- Promise对象对外提供了统一的回调函数接口（resolve和reject回调函数），在重写的代码中，我们可以很容易的把对web service的请求分装到一个模块中，从而对外隐藏Http Request的所有细节。
- 通过Promise对象的封装，对错误代码的处理被统一了，都是通过对 `reject` 函数调用来说明异步操作过程中有错误发生，而且错误被集中到 `.catch` 代码段进行了处理（错误都被输出到了控制台）。
- 通过Promise的封装，异步操作的代码变得和同步操作代码很像，更方便其他人理解代码的处理逻辑。

```
'use strict';
var http = require("http");
var fs = require("fs");
var querystring = require("querystring");
var postData = querystring.stringify({
  'msg' : 'Hello World!'
});

var options = {
  hostname: '127.0.0.1',
  port: 8080,
  path: '/upload',
  method: 'POST',
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded',
    'Content-Length': postData.length
  }
};

var pHttpRequest = new Promise(function(resolve, reject){
    let req = http.request(options, (res) => {
      console.log(`STATUS: ${res.statusCode}`);
      console.log(`HEADERS: ${JSON.stringify(res.headers)}`);
      res.setEncoding('utf8');

      let dataReceived = ""

      res.on('data', (chunk) => {
        dataReceived = dataReceived + chunk.toString();
      });

      res.on('end', () => {
        resolve(dataReceived);
      })
    });

    req.on('error', function(e){
        reject(e);
    });
```

```
    req.write(postData);
    req.end();
})

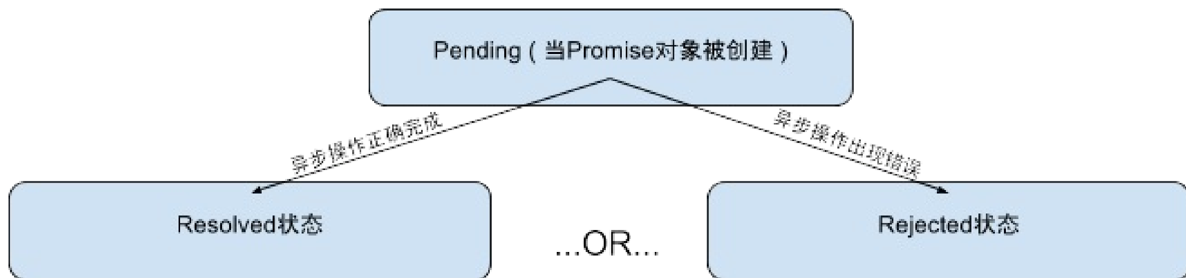
pHttpRequest.then(
    //http request promise成功时候的处理
    //在http request promise成功的时候，开始处理写文件操作
    function(dataReceived){
        return new Promise(function(resolve, reject){
            fs.writeFile("temp.txt", dataReceived, function(e){
                if (e) reject(e);
                else resolve("Write file succ");
            });
        })
    }
).then(
    //writeFile promise成功时候的处理
    function(msg){
        console.log(msg);
    }
).catch(
    //全局错误处理
    function(err){
       console.log("some error happen(" + err + ")");
    }
)
```

正式基于Promise对象有上面所说的这些优点，ES6正式将Promise对象编程了系统的一个内置对象，大家再也不用通过第三方库开始用Promise对象了。

Promise对象的特性

- Promise对象一旦创建，就开始执行了，你没有办法取消Promise对象的执行。
- Promise对象的状态只由异步操作的结果决定，没有任何其他的操作可以改变Promise对象的状态如果异步操作执行成功，Promise对象将进入Resolved状态，同时resolve函数将被调用；如果异步操作执行失败，Promise对象将进入Rejected状态，同时reject函数将被调用。
- Promise对象一旦进入Resolved或者Rejected状态，状态将不可能再发生变化，在Promise对象被销毁之前，将一直保持Resolved或者Rejected状态。
- 因为Promise对象的实现方法，在异步操作过程中出现的异常是不会被抛出的，因此需要在Promise对象内部进行处理（通过提供 `.catch` 代码段来实现）。

下图表示了一个Promise对象的整个生命周期。



Promise对象的使用

创建Promise对象

在ES6中，你可以通过 `new Promise(function(resolve, reject){ })` 来创建Promise对象，下面是一个具体的代码。

```
'use strict';
var fs = require("fs");
var promiseObj = new Promise(function(resolve, reject){
    //put some code to call
    fs.readFile("temp.txt", (err, data) => {
        if (err){
            reject(err);
        }else{
            resolved(data);
        }
    });
});
```

关联resolve function和reject function

在上面的例子中，如果你执行这个代码，你会发现没有任何的效果（没有输出），那是因为你没有给这个创建的Promise对象关联相应的resolve和reject函数。下面是一个进一步的例子，这个例子将给Promise对象绑定resolve和reject函数，你就可以看到效果了。在我的测试环境中，因为我们有名为"temp.txt"的文件存在，所以输出了 `file read fail` 。

```
'use strict';
var fs = require("fs");
var promiseObj = new Promise(function(resolve, reject){
    //put some code to call
    fs.readFile("temp.txt", (err, data) => {
        if (err){
            reject(err);
        }else{
            resolved(data);
        }
    });
});
promiseObj.then(function(data){
    console.log("file read succ");
}, function(err){
    console.log("file read fail");
})
```

在通常情况下，reject状态的函数我们一般不在then中设置，而是在catch中设置，这样代码看起来更像是传统意义上的同步代码(和try...catch比较）。因此上面的例子可以重新写成

```
'use strict';
var fs = require("fs");
var promiseObj = new Promise(function(resolve, reject){
    //put some code to call
    fs.readFile("temp.txt", (err, data) => {
        if (err){
            reject(err);
        }else{
            resolved(data);
        }
    });
});
promiseObj.then(function(data){
    console.log("file read succ");
}).catch(function(err){
    console.log("file read fail");
});
```

级联多个Promise对象

Promise对象的resolve函数的参数可以是另外一个Promise对象，这样就可以将2个Promise对象级联起来。下面是一个简单的例子

```
'use strict';
var p1 = new Promise(function(resolve, reject){
    setTimeout(() => reject(new Error("something test"), 3000));
});

var p2 = new Promise(function(resolve, reject){
    setTimeout(() => resolve(p1), 1000);
});

p2.then(function(data){
   console.log("p2 succ");
}).catch(function(err){
   console.log("p2 fail");
});
```

Promise的特殊函数

Promise.all()

将多个Promise包装成一个全新的Promise Object，如果所有的Promise被Resolved，那么新的Promise将被Resolve；否则新的Promise将被Reject。

Promise.race()

和.all一样，.race将把多个Promise包装成一个新的Promise Object，不同的地方是。这些Promise之中任何一个Resolve或者Reject了，新的Promise就被Resolve或者Reject了。

Promise.resolve()

将传入的对象封装成一个Promise对象返回。resolve方法根据以下的规则返回Promise对象。

- 如果输入的参数本身是一个Promise对象，那么resolve方法直接返回这个对象；
- 如果输入的对象本身有then方法（必须是一个可以接受2个function的方法），那么resolve将这个对象转换成Promise对象，并理解调用then方法；下面是一个例子

```
//for this example, you will see following output
//   then function in thenobject
//   Promise object resolve function is called Then function is cal
var thenobject = {
    then: function(resolve, reject){
       console.log("then function in thenobject");
       resolve("Then function is called");
    }
};

var pObj = Promise.resolve(thenobject);
pObj.then(function(data){
    console.log("Promise object resolve function is called " + data
});
```

- 如果传入的参数就是一个普通对象，那么返回的Promise对象直接处于resolved状态，并且输入的参数将作为resolved状态下调用的函数的参数。下面是一个具体的例子。

```
//for this example, you will see following output
//   Promise object resolve function is called Hello World!
var pObj = Promise.resolve("Hello World!");
pObj.then(function(data){
    console.log("Promise object resolve function is called " + data
});
```

*如果没有输入参数，那么返回的Promise对象直接处于resolved状态，并且resolved状态下调用的函数没有输入参数。

done() method

通过在Promise的调用链最后使用这个方法，可以保证catch到任何的错误。

finally() method

如果你需要在Promise结束的时候（不管resolve还是reject结束），都有一个函数被调用，那么就需要使用这个方法。这个方法接受一个回调函数作为输入。当Promise结束的时候，这个回调函数将被调用。

概述

在ES5中，所有的属性名使用的都是标准的字符串，如果你想修改一个别人提供的对象，并且为这个对象增加一个方法的时候，你就要非常小心了，因为你可能选择了一个已经存在的方法的名字。因此在ES6中引入了Symbol这个类型，当你使用Symbol类型来定义类的属性或者方法名的时候，ES6将保证这个属性和方法名称是全局唯一的。

创建Symbol实例

```
# 第一种方法直接使用Symbol构造函数
var s1 = Symbol();
# 为创建的Symbol对象指定一个名称
var s2 = Symbol("test");
# 使用这种方法，每次创建出来的Symbol对象都是不一样的
var s3 = Symbol("test");
# console.log(s2 == s3)将返回false

# 第二种方法，使用Symbol.for方法来创建Symbol实例
# 使用Symbol.for方法来创建一个Symbol实例的时候，
# 系统首先会在一个全局的注册表中查找是否有相同Key名称的Symbol被创建了，如果找
# 否则，将创建一个全新的对象
s1 = Symbol.for("test")
s2 = Symbol.for("test")
console.log(s1 == s2) //will return true
console.log(s1 == s3) // will return false，因为s3是通过Symbol调用产生
```

在我们了解了ES6为什么需要引入Symbol对象之后，让我们来看一些具体的例子来增加一些对Symbol对象的感性的认识。

引用实例1 --- 对象的属性

使用Symbol来定义类的属性

```
var a = {};
var s1 = Symbol("test");
a[s1] = "hello world!";
console.log(a[s1]); //this will print "Hello World!"
```

需要注意的是，当使用**Symbol**作为属性名的时候，一定要用**[]**来进行引用，如果你用**.**（点号）进行引用，那么产生的属性名实际是一个字符串，和**Symbol**对象本身没一点关系

遍历使用**Symbol**来定义的属性

因为Symbol对象不是一个字符串，所以原先的**Object.getOwnPropertyNames()**方法并不会返回它们，你需要使用专门的**Object.getOwnPropertySymbols()**方法来访问它们。

```
var a = {};
var s1 = Symbol("test");
var s2 = Symbol("testfunc");
a[s1] = "hello world!";
a[s2] = function(){
    console.log("Test function");
};
//below code will return []
console.log(Object.getOwnPropertyNames(a));
//below code will return [Symbol(test), Symbol(testfunc)]
console.log(Object.getOwnPropertySymbols(a));
```

引用实例**2 ---** 消除魔术字符串

魔术字符串指的是在代码中反复出现的相同的字符串，对一个结构良好的代码来说，应该尽量消除魔术字符串的存在，而使用常量作为替代。下面就让我们来看一个具体的例子来了解如何使用Symbol来消除代码中的魔术字符串。 在代码中我们尝试根据输入信息的不同返回不同的Hello

```
# 引入Symbol前的代码
# 在这个代码中es, ch就是魔术字符串
# 当你在调用这个函数的时候，必须确保输入的参数是一个有效的魔术值
function getHello(country){
    switch(country){
        case "es":
            return "Holla";
        case "ch":
            return "你好";
        default:
            return "Hello";
    }
}
console.log(getHello("es"));
console.log(getHello("ch"));
```

```
# 引入Symbol之后的代码
# 在这个代码中es, ch不再是一个具体的字符串
# 因此不管在COUNTRY_CODE中如何修改es, ch的定义，调用方的代码都不需要修改
const COUNTRY_CODE= {
    es: Symbol(),
    ch: Symbol()
}
function getHello(country){
    switch(country){
        case COUNTRY_CODE.es:
            return "Holla";
        case COUNTRY_CODE.ch:
            return "你好"
        default:
            return "Hello"
    }
}
console.log(getHello(COUNTRY_CODE.es));
console.log(getHello(COUNTRY_CODE.ch));
```

ES6内置的Symbol实例

Symbol.hasInstance

对对象使用instanceof调用的时候，在ES6内部调用的是obj[Symbol.hasInstance]方法

Symbol.isConcatSpreadable

决定调用Array.prototype.contact方法的时候，对象是否可以展开

```
var arr1 = [1, 2];
[3, 4].concat(arr1); // return [3, 4, 1, 2]

arr1[Symbol.isConcatSpreadable] = false;
[3, 4].concat(arr1); // return [3, 4, [1, 2]]
```

和字符串操作相关的Symbol

- Symbol.replace指向一个方法，将被String.prototype.replace调用
- Symbol.search指向一个方法，将被String.prototype.search调用
- Symbol.split指向一个方法，将被String.prototype.split调用
- Symbol.match指向一个方法，将被str.match(object)调用
- Symbol.toStringTag只想一个方法，将被Object.prototype.toString调用

Symbol.iterator

指向对象的默认遍历器

Symbol.toPrimitive

指向一个方法，将对象转换成一个primitive类型的值。这个方法将接受一个hint参数，表示要转换成哪种primitive类型，具体的value包括"Number", "String", "Default"

Symbol.unscopables

指向一个属性，这个属性返回一个JSON对象，表示该对象的哪些属性将被with环境排除在外，下面是一个具体的例子。

```
# when Test has no Symbol.unscopables
class Test{
saySomething() { return "abc"; }
}
var saySomething = function() { return "123"; }
with(Test.prototype){
saySomething(); // return "abc"
}
# when Test has Symbole.unscopables
class Test{
saySomething() { return "abc"; }
get [Symbol.unscopables] {
    return {saySomething: true};
}
}
with(Test.prototype){
saySomething(); // now return "123"
}
```

koa基础

- 官网（英文）：http://koajs.com/
- 官网（中文）：http://koa.bootcss.com/

上下文

koa的中间件

```
app.use(function *(next){
  this; // is the Context
  this.request; // is a koa Request
  this.response; // is a koa Response
});
```

说明：

- this是上下文(注释1*)
- *代表es6里的generator

http模型里的请求和响应

- this.request
- this.response

对比express的中间件

```
app.use(function (req, res, next) {
  return next();
});
```

express里的req和res是显式声明，看起来更清晰一些

next处理是一样的，二者无差异

注释1： 此处的this 并不同于通常状态下的this 指向（即调用者）。在koa中 this 指向每一次的请求，在请求接受后初始化，在一次请求结束后被释放。

koa-generator

这里的generator是生成器的意思，用于生成项目骨架，express-generator就是一个比较好的例子，虽然比较精简，但结构清晰，足矣满足一般性的开发需求

鉴于很多人非常熟悉expressjs，这里假定大家也熟悉express-generator

express-generator提供的功能

- 生成项目骨架
- 约定目录结构（经典，精简，结构清晰）
- 支持css预处理器

koa-generator提供的功能

- 生成项目骨架
- 约定目录结构（和express-generator的结构一模一样）
- 支持css预处理器（暂未实行）

2个生成器共同的项目骨架结构

- app.js为入口
- bin/www为启动入口
- 支持static server，即public目录
- 支持routes路由目录
- 支持views视图目录
- 默认jade为模板引擎

koa-generator支持koa1.x和2.x，安装后，可以分别使用 `koa` 和 `koa2` 分别创建。

安装koa-generator

```
$ npm install -g koa-generator
```

创建项目

koa-generator支持Koa1.x和2.x，安装后，可以分别使用 `koa` 和 `koa2` 分别创建。

Koa 1.x

```
$ koa helloworld
```

Koa 2.x

```
$ koa2 helloworld
```

切换视图模板引擎

视图默认使用的是 `jade` 。如果想使用其他的视图

```
$ koa 1.x/views-ejs -e
```

说明

- `-e, --ejs add ejs engine support (defaults to jade)`

koa-generator使用的是koa-views，支持所有consolidate.js支持模板引擎

路由

写法说明

Koa 1.x

只要是koa-router写的路由都可以加载的，加载方式和express里一样

```
var router = require('koa-router')();

router.get('/', function *(next) {
  this.body = 'this /1!';
});


router.get('2', function *(next) {
  this.body = 'this /2!';
});

module.exports = router;
```

一定要区分

```
url = /2
router.get('2', function *(next) {
  this.body = 'this /2!';
});
```

```
url = //2
router.get('/2', function *(next) {
  this.body = 'this /2!';
});
```

这个是koa-router的一个问题，和express里的路由稍有不一样，注意一些即可

Koa 2.x

由于Koa 2.x支持async，故写法稍有差异

```
var router = require('koa-router')();

router.get('/', async function (ctx, next) {
  await ctx.render('index', {
    title: 'Hello World Koa!'
  });
});
module.exports = router;
```

HTTP

Get

```
npm run 1
```

如何获取query参数

routes/index.js

```
var router = require('koa-router')();

router.get('/', function *(next) {
  console.log(this.request.query)
  console.log(this.query)

  yield this.render('index', {
    title: 'Hello World Koa!'
  });
});

module.exports = router;
```

访问http://127.0.0.1:3000/?a=1

日志

```
  <-- GET /?a=1
{ a: '1' }
{ a: '1' }
```

和express里获取query的方法是一样的，req.query

koa里是

- this.request.query
- this.query

这里需要说明以下this上下文上有request和response2个对象，每次写起来又比较麻烦

于是把request和response上的方法也丢给this，这样就相当于this上有了对应request和response里的方法的别名（简写方式）

- 别名列表

Request aliases

以下访问器和别名与 Request 等价：

- ctx.header
- ctx.method
- ctx.method=

- ctx.url
- ctx.url=
- ctx.originalUrl
- ctx.path
- ctx.path=
- ctx.query
- ctx.query=
- ctx.querystring
- ctx.querystring=
- ctx.host
- ctx.hostname
- ctx.fresh
- ctx.stale
- ctx.socket
- ctx.protocol
- ctx.secure
- ctx.ip
- ctx.ips
- ctx.subdomains
- ctx.is()
- ctx.accepts()
- ctx.acceptsEncodings()
- ctx.acceptsCharsets()
- ctx.acceptsLanguages()
- ctx.get()

Response aliases

以下访问器和别名与 Response 等价：

- ctx.body
- ctx.body=
- ctx.status
- ctx.status=
- ctx.length=
- ctx.length
- ctx.type=
- ctx.type
- ctx.headerSent
- ctx.redirect()
- ctx.attachment()
- ctx.set()
- ctx.remove()
- ctx.lastModified=
- ctx.etag=

如何获取params

express里经典用法

http://expressjs.com/en/4x/api.html#app.param

```
app.get('/user/:id', function (req, res, next) {
  console.log('although this matches');
  next();
});
```

请求是

访问http://127.0.0.1:3000/users/alfred

那么koa里如何使用呢？

关于路由

- express是自带路由
- koa这货没有，所以，需要另外集成，koa-generator使用的是目前比较流行的koa-router（我喜欢它的是Express-style）

https://github.com/alexmingoia/koa-router

好吧

routes/users.js

```
var router = require('koa-router')();

router.get('/:id', function *(next) {
  console.log(this.params);
  console.log(this.request.params);
  this.body = 'this a users response!';
});

module.exports = router;
```

执行

```
npm run 2
```

访问http://127.0.0.1:3000/users/alfred

日志

```
  <-- GET /users/alfred
{ id: 'alfred' }
undefined
GET /users/alfred - 28
```

首先肯定一点，this.params是可以取到params的，这点和express路由用法类似

但是注意的是

```
this.request.params != this.params
```

这说明params不是request上的方法，翻查源码，确实是如此

https://github.com/alexmingoia/koa-router/blob/5.x/lib/router.js#L317

Post

todo

从post获取参数

Query参数

同Get里如何获取query参数

Params参数

同Get里如何获取params参数

标准表单 Post with x-www-form-urlencoded

see `public/post.html`

```
<script>
$(function(){
    $.ajaxSetup({
      contentType: "application/x-www-form-urlencoded; charset=utf-
    });

    $.post("/users/post", { name: "i5a6", time: "2pm" },
       function(data){
         console.log(data);
       }, "json");

});
</script>
```

in `routes/users.js`

```
router.post('/post', function(req, res) {
  // res.send('respond with a resource');
    res.json(req.body);
});
```

测试

```
$ npm test
```

使用Postman测试

Normal | Basic Auth | Digest Auth | OAuth 1.0 | No environment

http://127.0.0.1:3001/users/post POST

form-data | x-www-form-urlencoded | raw

Key	Value
a	1
b	2
Key	Value
Key	Value

Send | Preview | Add to collection

Body | Cookies (6) | Headers (5) | STATUS 200 OK | TIME 32 ms

Pretty | Raw | Preview | JSON | XML

```
1 {
2     "a": "1",
3     "b": "2"
4 }
```

文件上传 Post with form-data

主要目的是为了上传

koa-v1 要是用 koa-multer-v0.0.2 对应的 multer < 1，所以本处需要指定版本安装

```
$ npm install --save koa-multer@0.0.2
```

Usage

```
var app = require('koa')()
  , koa = require('koa-router')()
  , logger = require('koa-logger')
  , json = require('koa-json')
  , views = require('koa-views')
  , onerror = require('koa-onerror');


var multer = require('koa-multer');

app.use(multer({ dest: './uploads/'}));
```

You can access the fields and files in the request object:

```
router.post('/post/formdata', function *(next) {
  console.dir(this.req.body)
  console.dir(this.req.files)

  this.body = 'this a users response!';
});
```

重要提示： Multer will not process any form which is not multipart/form-data

see more

测试

```
$ npm test
```

使用Postman测试

Normal | Basic Auth | Digest Auth | OAuth 1.0 | No environment

http://127.0.0.1:3001/users/post/formdata POST

form-data | x-www-form-urlencoded | raw

a	1	Text
b	2	Text
pic	Choose Files 12.pic.jpg	File
Key	Value	Text

Send | Preview | Add to collection

Body | Cookies (6) | Headers (5) | STATUS 200 OK | TIME 46 ms

Pretty | Raw | Preview | JSON | XML

```
1 {
2     "a": "1",
3     "b": "2"
4 }
```

Post with raw(todo)

To get the raw body content of a request with Content-Type: "text/plain" into req.rawBody you can do:

https://gist.github.com/tj/3750227

req.rawBody已经被干掉了，现在只能用req.text

下面是tj给出的代码片段

```
var express = require('./')
var app = express();

app.use(function(req, res, next){
  if (req.is('text/*')) {
    req.text = '';
    req.setEncoding('utf8');
    req.on('data', function(chunk){ req.text += chunk });
    req.on('end', next);
  } else {
    next();
  }
});

app.post('/', function(req, res){
  res.send('got "' + req.text + '"');
});

app.listen(3000)
```

测试

```
$ npm test
```

使用Postman测试

Normal | Basic Auth | Digest Auth | OAuth 1.0 | No environment

http://127.0.0.1:3001/users/post/raw POST

form-data | x-www-form-urlencoded | raw | JSON

```
1 {
2   "a": "1",
3   "b": "2",
4   "c": {
5     "a": "1",
6     "b": "2"
7   }
8 }
```

Send | Preview | Add to collection

Body | Cookies (6) | Headers (5) | STATUS 200 OK | TIME 38 ms

Pretty | Raw | Preview | JSON | XML

```
1 {
2     "a": "1",
3     "b": "2",
4     "c": {
5         "a": "1",
6         "b": "2"
7     }
8 }
```

db

封装思路

koa依赖co，其中间件对非阻塞异步代码的要求必须是Yieldables列表中的形式，而mysql库是回调函数的形式。因此，我们需要进行封装，使其接口符合要求。

目前我找到了四种方法，前三种使用开源库，第四种自己动手，将express下的dbHelper层封装成co最新支持的Promise形式。

实现方法一（co-mysql）

co-mysql和mysql-co实现了对mysql或mysql2的封装转化。这两个库的思路差不多，mysql-co封装度更高，并使用速度更快的mysql2；而co-mysql更简单，只是将mysql.query封装成Promise形式。下面是基于co-mysql的示例代码：

```
var wrapper = require('co-mysql'),
  mysql = require('mysql');
var options = { /* 数据库连接字串 */ };

var pool = mysql.createPool(options),
  p = wrapper(pool);

...
  var rows = yield p.query('SELECT 1');
  yield this.render('index', {
        title: rows[0].fieldName
    });
...
})();
```

实现方法二（promisify-node）

使用promisify-node库，可以将库整体转化为Promise形式，也可以方便的转化库中的指定接口函数，示例代码如下：

```
var promisify = require("promisify-node");
var db = promisify("dbHelper");    //express下的回调形式封装库
...
var rows = yield db.getById('tableName', {id:1});
    yield this.render('index', {
        title: rows[0].fieldName
    });
...
```

实现方法三（thunkify）

使用thunkify也能够完成封装，thunkify-wrap是一个增强版的thunkify。不过看说明，这种方法在未来的发展中可能会被淘汰，大概的使用方法如下：

```
var genify = require('thunkify-wrap').genify;
var db = genify("dbHelper");     //express下的回调形式封装库
...
var rows = yield db.getById('tableName', {id:1});
    yield this.render('index', {
        title: rows[0].fieldName
    });
...
```

实现方法四（直接方法）

直接改造原来express下的回调方式代码为Promise形式，代码及说明如下：

dbHelper.js

```
var options = { /* 数据库连接字串 */ };

var mysql = require('mysql');
var pool = mysql.createPool(options);

//原有非接口代码，对mysql的封装，执行sql语句
function execQuery(sql, values, callback) {
    var errinfo;
    pool.getConnection(function(err, connection) {
        if (err) {
            errinfo = 'DB-获取数据库连接异常！';
            throw errinfo;
        } else {
            var querys = connection.query(sql, values, function(err
                release(connection);
                if (err) {
                    errinfo = 'DB-SQL语句执行错误:' + err;
                    callback(err);
                } else {
                    callback(null,rows);        //注意：第一个参数必须
                }
            });
        }
    });
}

function release(connection) {
    try {
        connection.release(function(error) {
            if (error) {
                console.log('DB-关闭数据库连接异常！');
            }
        });
    } catch (err) {}
}
//对外接口代码，包装成返回Promise函数的形式
exports.getById = function(tablename, id){
    return new Promise(function(resolve, reject){
        var values = {id:id};
        var sql = 'select * from ?? where ?';
        execQuery(sql,[tablename, values], function(err, rows){
            if(err){
                reject(err);
            }else{
                resolve(rows);
            }
        })
    });
}
```

routes/index.js

```
var db = require("../dbHelper");
...
var rows = yield db.getById('tableName', {id:1});
    yield this.render('index', {
        title: rows[0].fieldName
    });
...
```

代码

示例部分代码取自该项目：

```
https://github.com/zhoutk/koadmin.git
```

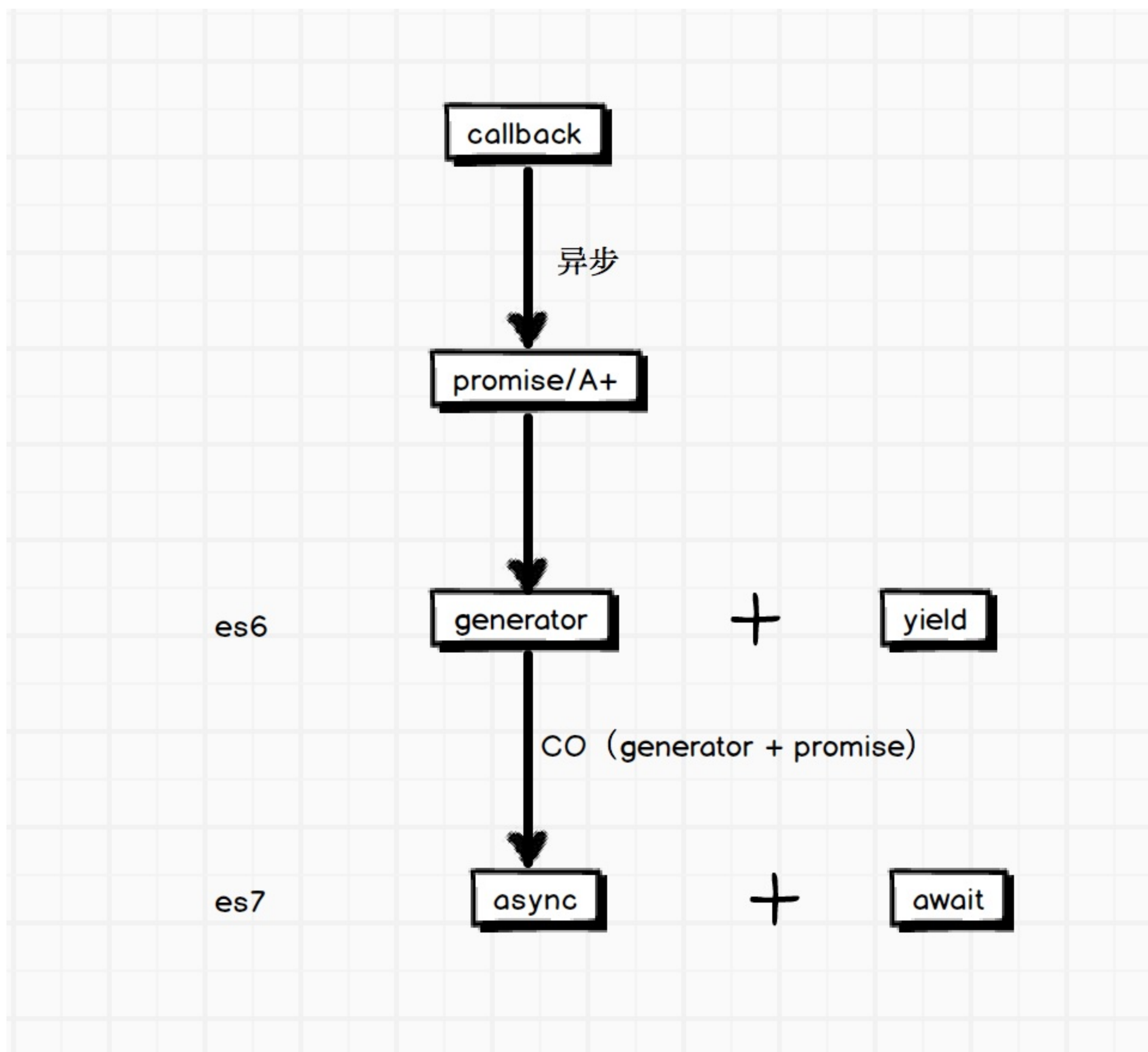
小结

koa框架以co库为核心组织，很好的用generator来解决了回调函数问题。进行Promise接口形式包装的时候，要注意：回调函数要完全符合其要求的形式：

```
function(err, rows){
            if(err){
                reject(err);
            }else{
                resolve(rows);
            }
        })
```

mongodb

流程控制



http://www.ruanyifeng.com/blog/2015/05/async.html

generator/co

es6的generator是什么？

generator指的是

```
function* xxx(){
}
```

是es6里的写法。

```
function* test() {
    console.log('1');
    yield 1;
    console.log('2');
    yield 2;
    console.log('3');
}
```

代码中间插了两行yield，代表什么呢？

- 当test执行到 yield 1这一行的时候，程序将被挂起，要等待执行下一步的指令；
- 当接收到指令后，test将继续往下运行，直到yield 2这一行，然后程序又被挂起并等待指令；
- 收到指令后，test又将继续运行，而下面已经没有yield了，那么函数运行结束。

这是不是就像，我们调试代码的时候，给插的断点 ？

当然，断点这个比喻，只是表象上比较相像，实质原理还是有非常大差异。

yield就是让后面的generator执行完成后，才继续往下走。

要注意，function后面多了一个星号，这样是表明这个函数将变成一个生成器函数，而不是一个普通函数了。意思就是，test这个函数，将不能被这样执行

test(); 但可以获得一个生成器

var gen = test(); // gen就是一个生成器了 然后，生成器可以通过next()来执行运行

gen.next();

也就是上面说的，让函数继续运行的指令。

简单地总结一下：

- 生成器通过yield设置了一些类似”断点“的东西，使得函数执行到yield的时候会被阻断；

- 生成器要通过next()指令一步一步地往下执行（两个yield之间为一步）；
- yield 语句后面带着的表达式或函数，将在阻断之前执行完毕；
- yield 语句下面的代码，将不可能在阻断之前被执行；

由此可以看出，yield是如何将异步非阻塞代码，变成 异步阻塞代码。

co

理解了co的核心代码就理解了koa的流程控制

```
var ctx = this;
var args = slice.call(arguments, 1);
```

一开始保存上下文,把arguments的length属性去掉,剩余的参数转数组就是gen的参数

再来看return的promise内的代码

```
if (typeof gen === 'function') gen = gen.apply(ctx, args);
if (!gen || typeof gen.next !== 'function') return resolve(gen);
```

先判断gen是不是generator function,如果是就转为generator,相当于 `gen = new gen;`

转为generator之后就可以调用gen.next()了;

```
onFulfilled();
```

这是进入循环调用链的入口

```
function onFulfilled(res) {
  var ret;
  try {
    ret = gen.next(res);
  } catch (e) {
    return reject(e);
  }
  next(ret);
}
function onRejected(err) {
  var ret;
  try {
    ret = gen.throw(err);
  } catch (e) {
    return reject(e);
  }
  next(ret);
}
function next(ret) {
  if (ret.done) return resolve(ret.value);
  var value = toPromise.call(ctx, ret.value);
  if (value && isPromise(value)) return value.then(onFulfilled, onR
  return onRejected(new TypeError('You may only yield a function, p
    + 'but the following object was passed: "' + String(ret.value)
}
```

ret是gen.next后的{value:"done:"}对象,value是yield后的表达式,done是执行状态.

判断ret.done是否为true来确定是否需要再执行下去.为true时,说明已经是generator的最后一步,promise转为resolve.不为true时,将yield后的表达式转化为promise.

先判断是否转化为了promise,转化成功,就通过 `value.then(onFulfilled, onRejected)` 执行onFulfilled或onRejected,再次调用next(),实现循环调用.

当value不能转为promise时,抛出错误,promise转为reject,停止继续运行.

下面写一个例子简单分析一下：

```
var co = require('co');
var fs = require('fs');
function thunkRead(name) {
  return function (cb) {
    fs.readFile(name, function (err, file) {
      cb(err, file);
    });
  }
}
co(function *() {
  var file = yield thunkRead("package.json");
  console.log(file);
  return file;
}).then(function (file) {
  console.log(file);
});
```

通过上面这段代码来看一下co的整个流程

先模拟一个名为thunkRead的thunk函数，再看co里面的代码，co里面是一个generator function， `gen = gen.apply(ctx, args);` 通过这一句转化为了generator。

再进入onFulfilled()函数，第一个gen.next()之后ret是 `{ value: [Function], done: false }` 将ret传入next()中，done为false，所以toPromise，value是function，所以thunkToPromise.

```
function thunkToPromise(fn) {
  var ctx = this;
  return new Promise(function (resolve, reject) {
    fn.call(ctx, function (err, res) {
      if (err) return reject(err);
      if (arguments.length > 2) res = slice.call(arguments, 1);
      resolve(res);
    });
  });
}
```

因为fn是thunk函数，参数只有一个回调函数, `fn.call(ctx, function (err, res) {});` 直接调用resolve,在上面的例子中是resolve(file);

然后回到next()中，此时已经是一个promise对象，调用value的then方法，onFulfilled的参数就是file，再运行gen.next(file)，将上一步yield的结果file传入generator，因为在例子中最后return了file, ret是 `{ value:<Buffer ...>, done: true }` 最后不返回值的话应该是 `{ value: undefined, done: true }` ，再进入到next()中， 此时done已经为true，说明已经是generator的最后一步，resolve(value);

co中的代码已经执行结束，因为co也是一个promise, 最后resolve(value)，所有可以在then方法中得到这个value.

补充下toPromise支持转化thunks,array,objects,generators,generator functions.

所以可以yieldable的是以下6种:

- promises
- thunks (functions)
- array (parallel execution)
- objects (parallel execution)
- generators (delegation)
- generator functions (delegation)

async/await

promise with bluebird

test

mocha

supertest

deploy

最佳实践

faq

gitbook发布说明

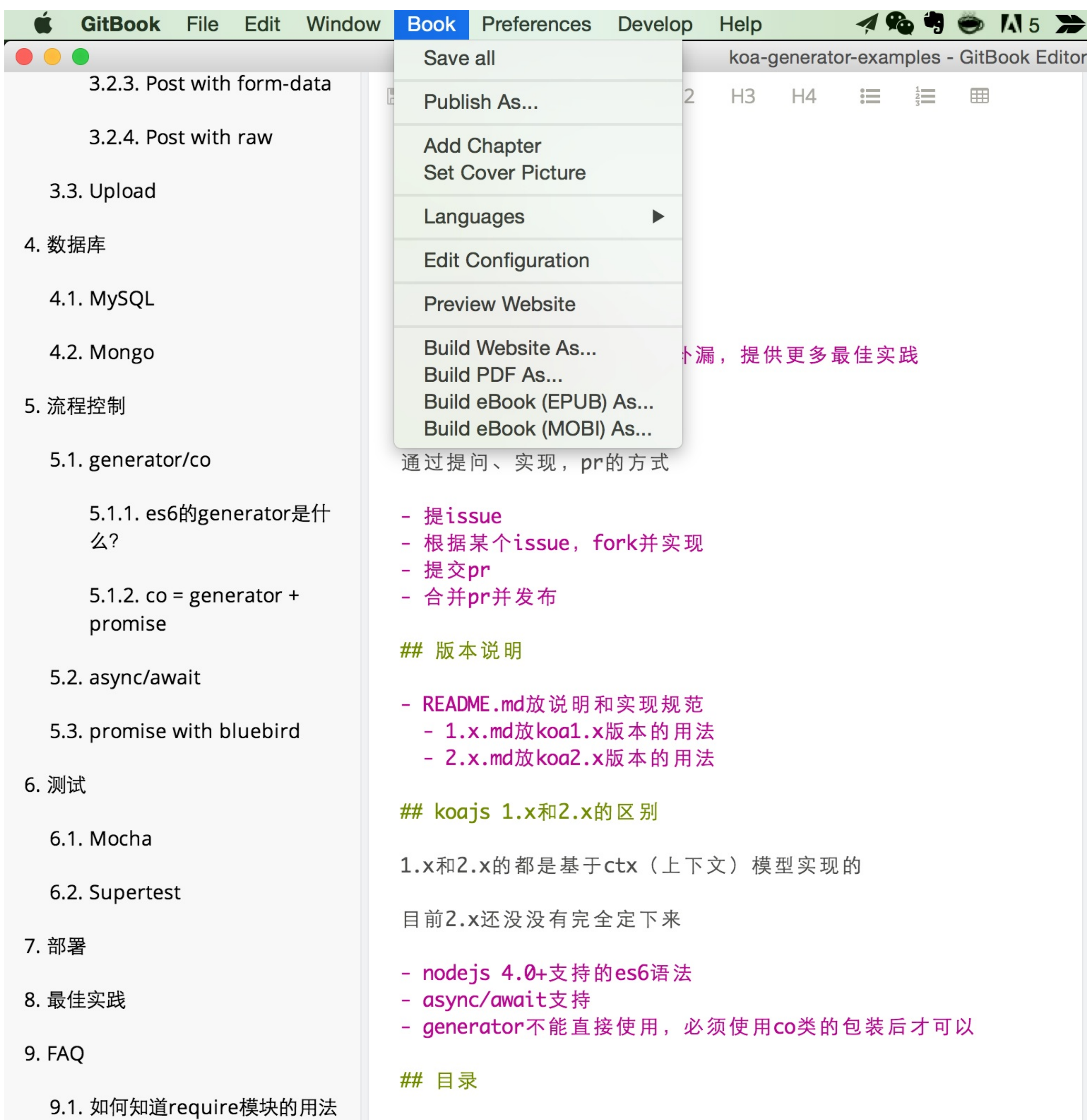
本书以gitbook为编辑器，发布到git pages上，为了便于大家使用，请按照如下办法生成并发布。

下载gitbook编辑器

https://www.gitbook.com/editor

生成html

> gitbook -> Book -> Preview Website



发布到git pages上

```
gulp deploy
```

预览并确认

访问 http://base-n.github.io/koa-generator-examples/

如何知道require模块的用法

比如

```
var router = require('koa-router')();
```

打开

https://www.npmjs.com/search?q=koa-router

找到

https://www.npmjs.com/package/koa-router

koa中的异常处理

比如router中某个yield可能抛错，这种情况下要返回请求是怎么做的呀，都是直接在app on error里面处理吗

在route里

```
co(function* () {
  var result = yield Promise.resolve(true);
  return result;
}).then(function (value) {
  console.log(value);
}, function (err) {
  // 异常处理
  console.error(err.stack);
});
```